FinancialDataAnalysis

Fundamentals

A good place to start is to define a one-period return calculation.

Objectives:

- Interactive programming
- How to calculate returns.
- Working with data.
- How to calculate return volatility

Interactive programming

We are going to focus on interactive programming. This is the most productive (and most common) type of analytic programming. In constrast to compiled programs (e.g, C, C++, Fortran, Java), interactive programs:

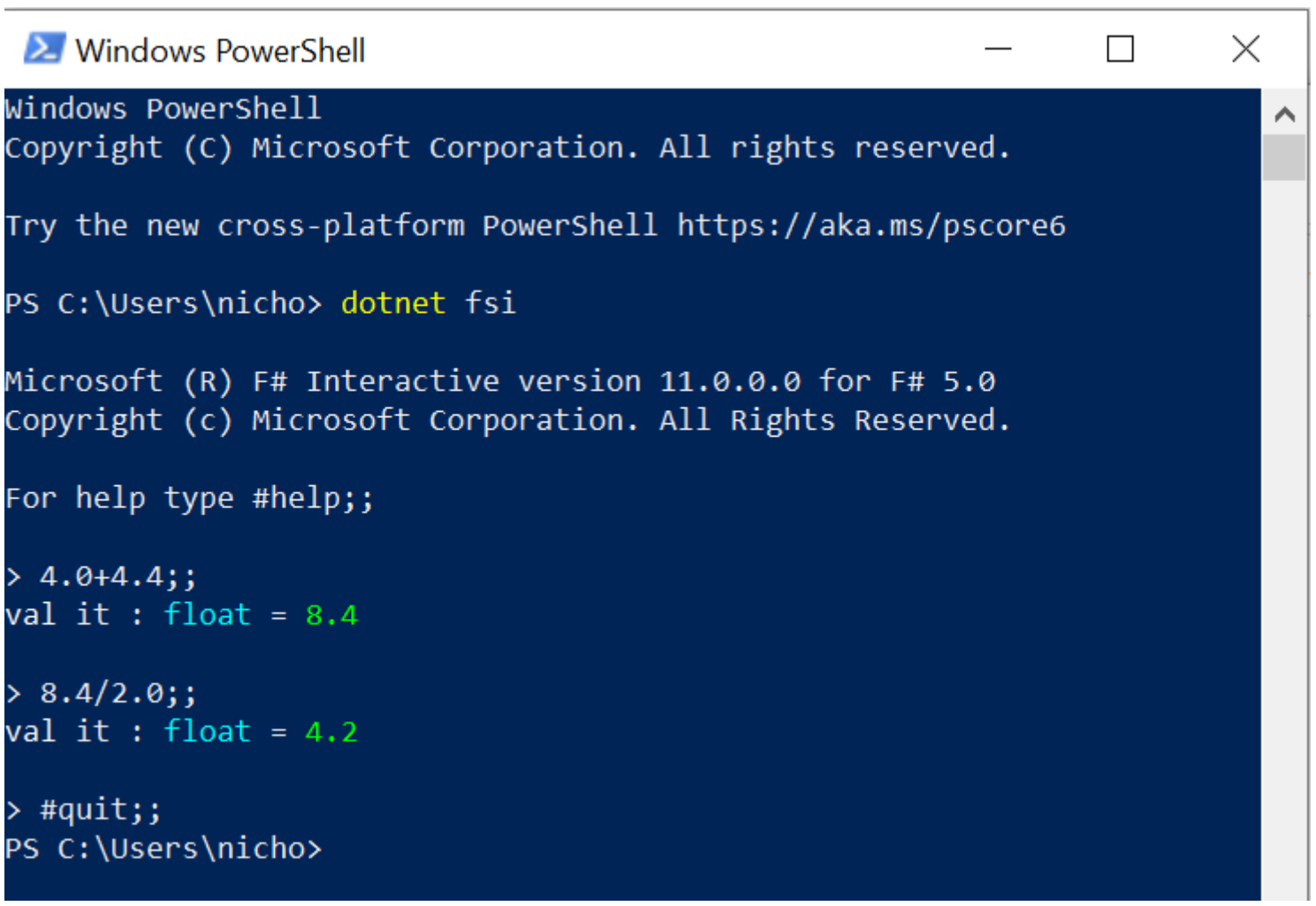
- Allow rapid iterative development.
- You can quickly quickly write and rewrite sections of code, evaluating the output, without having to rerun the entire program.
- This is especially useful for financial analysis, because we often evaluate large datasets that take a long time to process.

Interactive programming typically involves a REPL (Read, Evaluate, Print, Loop). It is common for scripting langauges such as R, Python, Julia, Ruby, and Perl.

The terminal

The most basic way that you can run interactive code is at the command line using an interpreter. We can star the F# interactive interpreter by opening a terminal (e.g., terminal.app, cmd, powershell) and running `dotnet fsi`.

Once fsi is open, we can type a code snippet in the prompt followed by ";;" to terminate it and it will run.



It is fine to run code this way, but we can do better using an IDE (Integrated development environment) that incorportes syntax highlighting, intellisense tooltips, and execution.

Calculating returns

Basic calculations in fsi

Let's assume that you have $120.00 today and that you had $100.00 a year ago. Your annual return is then:

```
(120.0 / 100.0) - 1.0
```

```
val it : float = 0.2
```

Basic numerical types: float, int, and decimal

Notice that I included zeros after the decimal point. This is important. The decimal point makes it a floating point number. Floating point numbers (floats) are the most commonly used numerical type for mathematical calculations.

If we left the decimal off it would be an integer and we would get the wrong answer because integers cannot represent fractions.

```
(120/100 ) - 1
```

```
val it : int = 0
```

The other numerical data type is decimal.

```
(120m/100m ) - 1m
```

```
val it : decimal = 0.2M
```

Decimals are used when you need an exact fractional amount. Floats are insufficient in these circumstances because *"... such representations typically restrict the denominator to a power of two ... 0.3 (3/10) might be represented as 5404319552844595/18014398509481984 (0.29999999999999988897769...)"* (see wiki).

Static type checking

Finally, since F# is staticly typed, we must do arithmetic using numbers that are all the same type. If we mix floats and integers we will get an error:

```
(120.0 / 100) - 1.0
```

```
fundamentals.fsx(18,10): error FS0001: The type 'int' does not match the type 'float'
```

Static tying can slow you down a bit writing simple small programs, but as programs get larger and more complex the benefits become apparent. Specifically, static typing as implemented by F#:

- helps you ensure that the code is correct (i.e., the type of the input data matches what the function expects). In the words of Yaron Minksy at Janestreet, you can "make illegal states unrepresentable" (see here for F# examples).
- it also facilitates editor tooling that can check your code without running it and give tooltip errors (you should have seen a tooltip error in your editor if you type `(120.0 / 100) - 1.0` in your program file). It's like clippy on steriods (you are too young, but your parents might get this reference).

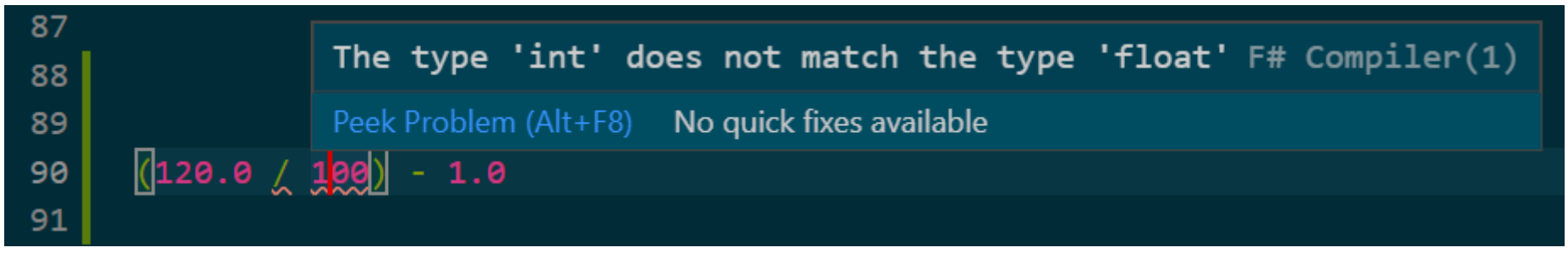


Image source

Assigning values

We could also do the same calculations by assigning $120.00 and $100.0 to named values.

```
let yearAgo = 100.0
let today = 120.0
(today / yearAgo) - 1.0
```

```
val yearAgo : float = 100.0
val today : float = 120.0
val it : float = 0.2
```

This works for one-off calculations, but if we want to do this more than once, then it makes more sense to define a function to do this calculation.

Defining functions

We can easily define a function to calcuate this return. Functions map (or transform) inputs into outputs.

```
let calcReturn pv fv = (fv / pv) - 1.0
```

```
val calcReturn : pv:float -> fv:float -> float
```

The type signature tells us that `calcReturn` is a function with two float inputs (pv and fv) and it maps those two inputs into a float output. The program was able to infer that `pv` and `fv` are floats because of the `1.0` float in the calculation.

We can execute it on our previously defined `yearAgo` and `today` values:

```
calcReturn yearAgo today
```

```
val it : float = 0.2
```

Or we can execute it on simple floats:

```
calcReturn 100.0 120.0
```

```
val it : float = 0.2
```

However, if we try to get it to execute on decimals, we will get an error because we defined the function to only operate on floats. This is another (simple) example of the compiler using type checking.

```
calcReturn 100.0m 120.0m
```

```
fundamentals.fsx(23,14): error FS0001: This expression was expected to have type
    'float'
but here has type
    'decimal'
```

Handling dividends

Our prior return calculation did not handle cash distributions such as dividends. We can incorporate dividends with a small modificaton:

```
let simpleReturn beginningPrice endingPrice dividend =
    // This is solving for `r` in FV = PV*(1+r)^t where t=1.
    (endingPrice + dividend) / beginningPrice - 1.0
```

The examples thus far have used simple (per period) compounding. We can also calculate continuously compounded returns, sldo known as log returns.

```
let logReturn beginningPrice endingPrice dividend =
    // This is solving for `r` in FV = PV*e^(rt) where t=1.
    log(endingPrice + dividend) - log(beginningPrice)
```

These two calculations give slightly different returns.

```
simpleReturn 100.0 110.0 0.0
```

```
val it : float = 0.1
```

```
logReturn 100.0 110.0 0.0
```

```
val it : float = 0.0953101798
```

It is typically not important which version of return you use so long as you are consistent and keep track of what type of return it is when you're compounding things.

Practice: Can you write a function to compound an initial investment of $100.00 at 6% for 5 years? You can calculate power and exponents using:

```
2.0**3.0
```

```
val it : float = 8.0
```

```
log 2.0
```

```
val it : float = 0.6931471806
```

```
exp 0.6931
```

```
val it : float = 1.999905641
```

```
exp(log(2.0))
```

```
val it : float = 2.0
```

Tuples

Looking at our return functions, we're starting to get several values that we're passing into the functions individaully. It can be useful to group these values together to make it easy to pass them around. Tuples are a simple way to group values.

```
(1,2)
```

```
val it : int * int = (1, 2)
```

```
(1,2,3)
```

```
val it : int * int * int = (1, 2, 3)
```

Tubles can contain mixed types.

```
(1,"2")
```

```
val it : int * string = (1, "2")
```

We can also deconstruct tuples. We can use built-in convenience functions for pairs.

```
fst (1,2)
```

```
val it : int = 1
```

```
snd (1,2)
```

```
val it : int = 2
```

We can also deconstruct tuples using pattern matching.

```
let (a, b) = (1, 2)
```

```
val b : int = 2
val a : int = 1
```

```
let (c, d, e) = (1, "2", 3.0)
```

```
val e : float = 3.0
val d : string = "2"
val c : int = 1
```

Now redefining our simple return function to take a single tuple as the input parameter.

```
let simpleReturnTuple (beginningPrice, endingPrice, dividend) =
    // This is solving for `r` in FV = PV*(1+r)^t where t=1.
    (endingPrice + dividend) / beginningPrice - 1.0

simpleReturnTuple (100.0, 110.0, 0.0)
```

```
val simpleReturnTuple :
  beginningPrice:float * endingPrice:float * dividend:float -> float
val it : float = 0.1
```

```
let xx = (100.0, 110.0, 0.0)
simpleReturnTuple xx
```

```
val xx : float * float * float = (100.0, 110.0, 0.0)
val it : float = 0.1
```

Records

If we want more structure than a tuple, then we can define a record.

```
type RecordExample =
    { BeginningPrice : float
      EndingPrice : float
      Dividend : float }
```

And construct a value with that record type.

```
let x = { BeginningPrice = 100.0; EndingPrice = 110.0; Dividend = 0.0}
```

```
val x : RecordExample = { BeginningPrice = 100.0
                          EndingPrice = 110.0
                          Dividend = 0.0 }
```

Similar to tuples, we can deconstruct our record value `x` using pattern matching.

```
let { BeginningPrice = aa; EndingPrice = bb; Dividend = cc} = x
```

```
val cc : float = 0.0
val bb : float = 110.0
val aa : float = 100.0
```

We can also access individual fields by name.

```
x.EndingPrice / x.BeginningPrice
```

```
val it : float = 1.1
```

We can define a return function that operates on the `RecordExample` type explicitly:

```
let simpleReturnRecord1 { BeginningPrice = beginningPrice; EndingPrice = endingPrice; Dividend = dividend} =
    // This is solving for `r` in FV = PV*(1+r)^t where t=1.
    (endingPrice + dividend) / beginningPrice - 1.0
```

Or we can let the compiler's type inference figure out the input type.

```
let simpleReturnRecord2 x =
    // This is solving for `r` in FV = PV*(1+r)^t where t=1.
    (x.EndingPrice + x.Dividend) / x.BeginningPrice - 1.0
```

Or we can provide a type hint to tell the compiler the type of the input.

```
let simpleReturnRecord3 (x : RecordExample) =
    // This is solving for `r` in FV = PV*(1+r)^t where t=1.
    (x.EndingPrice + x.Dividend) / x.BeginningPrice - 1.0
```

All 3 can be used interchangably, but when you have many similar types a type hint may be necessary to make the particular type that you want explicit.

```
simpleReturnRecord1 x
```

```
val it : float = 0.1
```

```
simpleReturnRecord2 x
```

```
val it : float = 0.1
```

```
simpleReturnRecord3 x
```

```
val it : float = 0.1
```

Working with data

With this foundation, let's now try loading some data. We are going to obtain and process the data using an external F# library called FSharp.Data that makes the processing easier.

Namespaces

First, let's create a file directory to hold data. We are going to use built-in dotnet IO (input-output) libraries to do so.

```
// Set working directory to the this code file's directory
System.IO.Directory.SetCurrentDirectory(__SOURCE_DIRECTORY__)
// Now create cache directory one level above the working directory
let cacheDirectory = "../data-cache"
if not (System.IO.Directory.Exists(cacheDirectory))
then System.IO.Directory.CreateDirectory(cacheDirectory) |> ignore
```

This illustrates the library namespace hierarchy. If we want to access the function within the hierarchy without typing the full namespace repetitively, we can open it. The following code is equivalent.

```
open System.IO
Directory.SetCurrentDirectory(__SOURCE_DIRECTORY__)
let cacheDirectory = "../data-cache"
if not (Directory.Exists(cacheDirectory))
then Directory.CreateDirectory(cacheDirectory) |> ignore
```

API keys

We are going to request the data from the provider tiingo. Make sure that you are signed up and have your API token. An API (application programming interface) allows you to write code to communicate with another program. In this case we are goig to write code that requests stock price data from tiingo's web servers.

Once you have your api key, create a file called `secrets.fsx` and save it at the root/top level of your project folder. In `secrets.fsx`, assign your key to a value named `tiingoKey`. If you are using git, make sure to add `secrets.fsx` to your `.gitignore` file.

```
let tiingoKey = "yourSuperSecretApiKey"
```

We can load this in our interactive session as follows, assuming that `secrets.fsx` is located one folder above the current one in the file system.

```
#load "../secrets.fsx"
```

and we can access the value by typing

```
Secrets.tiingoKey
```

Using external libraries

We are now going to use **FSharp.Data** to create a web request to tiingo's API . The documentation for the request format is here and we can find the documentation for making the web request using **FSharp.Data** here.

FSharp.Data is an external library, so we need to download the library from the nuget package manager, which is the primary package manager for the dotnet ecosystem. We can see on the "F# Interactive" tabe on the nuget page for the library what code to use to download it and reference it in once step from an F# interactive session:

```
#r "nuget: FSharp.Data" // To use a specific version such as 3.3.3 we'd use "nuget: FSharp.Data, 3.3.3"
```

Now we are ready to create the web request and cache the results.

```
open System
open System.IO
open FSharp.Data


let tiingoSampleFile = Path.Combine(cacheDirectory,"tiingo-sample.csv")
// Only do the request if the file doesn't exist.
if not (File.Exists(tiingoSampleFile)) then
    Http.RequestString
            ( "https://api.tiingo.com/tiingo/daily/gme/prices", httpMethod = "GET",
                query   = [ "token", Secrets.tiingoKey;
                            "startDate", "2020-10-01";
                            "endDate", "2021-02-03";
                            "format","csv"],
                headers = [HttpRequestHeaders.Accept HttpContentTypes.Csv])
    |> fun x -> File.WriteAllText(tiingoSampleFile, x)
```

Pipelines and lambda expressions

This download code used pipelining and lambda functions, which are two important language features. Pipelines are created using the pipe operator (`|>`) and allow you to pipe the output of one function to the input of another. Lambda expressions allow you to create functions on the fly.

```
1.0 |> fun x -> x + 1.0 |> fun x -> x ** 2.0
```

```
val it : float = 4.0
```

Collections: Arrays, Lists, Sequences

To see the data returned by tiingo we can read the file.

```
let lines = File.ReadAllLines(tiingoSampleFile)
```

```
lines |> Array.take 5
```

```
val it : string [] =
  [|"date,close,high,low,open,volume,adjClose,adjHigh,adjLow,adjOpen,adjVolume,divCash,splitFactor";
    "2020-10-01,9.77,10.25,9.69,10.09,4554055,9.77,10.25,9.69,10.09,4554055,0.0,1.0";
    "2020-10-02,9.39,9.78,9.3,9.38,4340484,9.39,9.78,9.3,9.38,4340484,0.0,1.0";
    "2020-10-05,9.46,9.59,9.2502,9.44,2804969,9.46,9.59,9.2502,9.44,2804969,0.0,1.0";
    "2020-10-06,9.13,9.835,9.1,9.56,4535421,9.13,9.835,9.1,9.56,4535421,0.0,1.0"|]
```

When we look at the type signature of the lines from the file `val lines : string []`, it tells us that we have a string array, meaning an array in which each element of the array is a line from the file. Arrays are "zero indexed", meaning the 0th item is the first in the array. We can access the elements individually or use a range to access multiple together.

```
lines.[0]
```

```
val it : string =
  "date,close,high,low,open,volume,adjClose,adjHigh,adjLow,adjOpen,adjVolume,divCash,splitFactor"
```

```
lines.[0 .. 2]
```

```
val it : string [] =
  [|"date,close,high,low,open,volume,adjClose,adjHigh,adjLow,adjOpen,adjVolume,divCash,splitFactor";
    "2020-10-01,9.77,10.25,9.69,10.09,4554055,9.77,10.25,9.69,10.09,4554055,0.0,1.0";
    "2020-10-02,9.39,9.78,9.3,9.38,4340484,9.39,9.78,9.3,9.38,4340484,0.0,1.0"|]
```

A simple float array.

```
let arr = [| 1.0 .. 10.0 |]
arr.[0]
arr.[0 .. 5]
```

```
val arr : float [] = [|1.0; 2.0; 3.0; 4.0; 5.0; 6.0; 7.0; 8.0; 9.0; 10.0|]
val it : float [] = [|1.0; 2.0; 3.0; 4.0; 5.0; 6.0|]
```

Lists and sequences are similar.

```
// List
[ 1.0 .. 10.0 ]
// Sequence
seq { 1.0 .. 10.0 }
```

Arrays, lists, and sequences have different properties that can make one data structure preferable to the others in a given setting. We'll discuss these different properties in due time, but for an overview you can see the F# collection language reference here. Sequences are the most different as they are "lazy", meaning "Sequences are particularly useful when you have a large, ordered collection of data but don't necessarily expect to use all the elements. Individual sequence elements are computed only as required, so a sequence can perform better than a list if not all the elements are used" (see F# language reference).

These collections have several built-in functions for operating on them such as map, filter, groupBy, etc.

```
arr
|> Array.map(fun x -> x + 1.0)
```

```
val it : float [] = [|2.0; 3.0; 4.0; 5.0; 6.0; 7.0; 8.0; 9.0; 10.0; 11.0|]
```

```
arr
|> Array.filter(fun x -> x < 5.0)
```

```
val it : float [] = [|1.0; 2.0; 3.0; 4.0|]
```

```
arr
|> Array.groupBy(fun x -> x < 5.0)
|> Array.map(fun (group, xs) -> Array.min xs, Array.max xs)
```

```
val it : (float * float) [] = [|(1.0, 4.0); (5.0, 10.0)|]
```

FSharp.Data Csv Type Provider

We're now going to process our downloaded data using the **FSharp.Data** Csv Type Provider. This is code that automatically defines the types of input data based on a sample. We have already reference the nuget packaged and opened the namespace, so we can just use it now.

```
let [<Literal>] tiingoSampleFileFullPath = __SOURCE_DIRECTORY__ + "/../data-cache/tiingo-sample.csv"
type TiingoCsv = CsvProvider<tiingoSampleFileFullPath>
let tiingoSample = TiingoCsv.GetSample()

tiingoSample.Rows
|> Seq.map(fun x -> x.Date.ToShortDateString(), x.Close)
|> Seq.take 5
|> Seq.toList
```

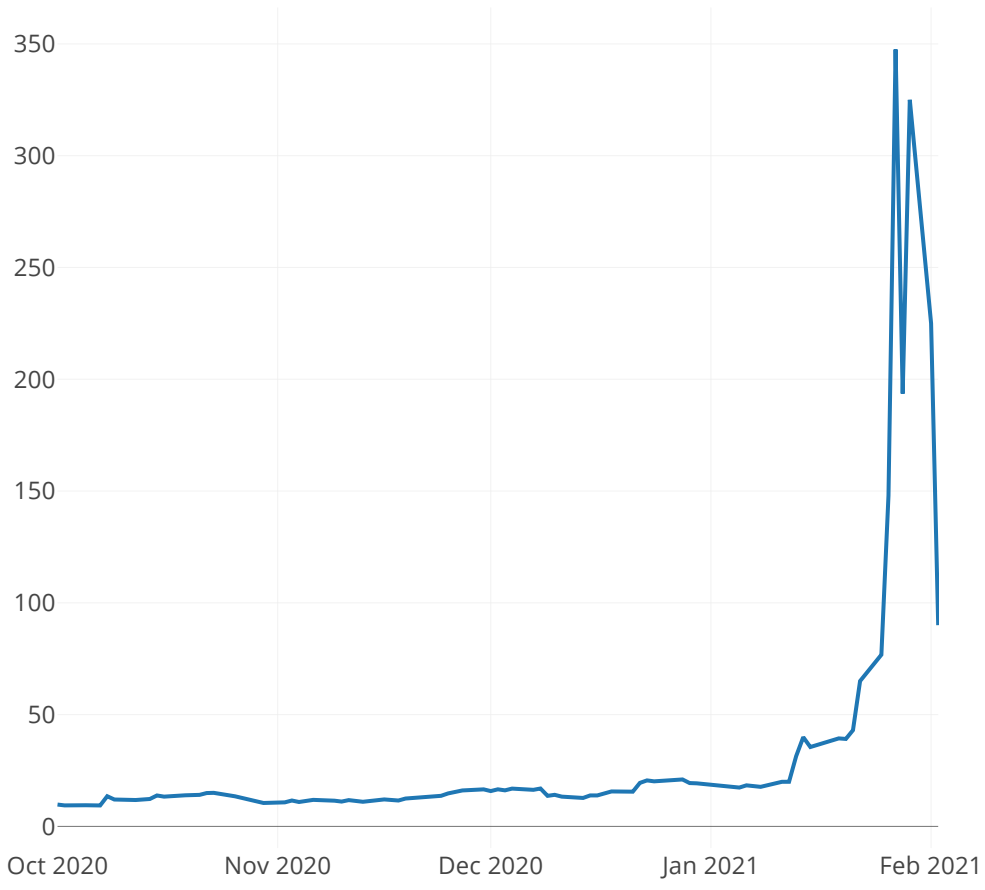
```
val tiingoSampleFileFullPath : string =
  "C:\Users\nicho\Dropbox\Research\FinancialDataAnalysis\1-Funda"+[39 chars]
type TiingoCsv = CsvProvider<...>
val tiingoSample : CsvProvider<...>
val it : (string * decimal) list =
  [("10/1/2020", 9.77M); ("10/2/2020", 9.39M); ("10/5/2020", 9.46M);
   ("10/6/2020", 9.13M); ("10/7/2020", 9.36M)]
```

Plotting

Now let's plot the stock price using Plotly.NET.

```
#r "nuget: Plotly.NET, 2.0.0-beta5"
open Plotly.NET


let sampleChart =
    tiingoSample.Rows
    |> Seq.map(fun x -> x.Date, x.AdjClose)
    |> Chart.Line
```

```
sampleChart |> Chart.Show
```



Let's calculate returns for this data. Typically we calculate close-close returns. Looking at the data, we could use the `close`, `divCash`, and `splitFacor` columns to calculate returns accounting for stock splits and dividends (a good at home exercise). But there is also an `adjClose` column that accounts for both those things. So we we can use this

```
// Returns
let returns =
    tiingoSample.Rows
    |> Seq.sortBy(fun x -> x.Date)
    |> Seq.pairwise
    |> Seq.map(fun (a,b) -> b.Date, calcReturn (float a.AdjClose) (float b.AdjClose))

let avgReturnEachMonth =
    returns
    |> Seq.groupBy(fun (date, ret) -> DateTime(date.Year, date.Month,1))
    |> Seq.map(fun (month, xs) -> month, Seq.length xs, xs |> Seq.averageBy snd)
```

We can look at a few of these

```
avgReturnEachMonth |> Seq.take 3 |> Seq.toList
```

```
val it : (DateTime * int * float) list =
  [(10/1/2020 12:00:00 AM {Date = 10/1/2020 12:00:00 AM;
                           Day = 1;
                           DayOfWeek = Thursday;
                           DayOfYear = 275;
                           Hour = 0;
                           Kind = Unspecified;
                           Millisecond = 0;
                           Minute = 0;
                           Month = 10;
                           Second = 0;
                           Ticks = 637371072000000000L;
                           TimeOfDay = 00:00:00;
                           Year = 2020;}, 21, 0.008618813543);
   (11/1/2020 12:00:00 AM {Date = 11/1/2020 12:00:00 AM;
                           Day = 1;
                           DayOfWeek = Sunday;
                           DayOfYear = 306;
                           Hour = 0;
                           Kind = Unspecified;
                           Millisecond = 0;
                           Minute = 0;
                           Month = 11;
                           Second = 0;
                           Ticks = 637397856000000000L;
                           TimeOfDay = 00:00:00;
                           Year = 2020;}, 20, 0.02443997661);
   (12/1/2020 12:00:00 AM {Date = 12/1/2020 12:00:00 AM;
                           Day = 1;
                           DayOfWeek = Tuesday;
                           DayOfYear = 336;
                           Hour = 0;
                           Kind = Unspecified;
                           Millisecond = 0;
                           Minute = 0;
                           Month = 12;
                           Second = 0;
                           Ticks = 637423776000000000L;
                           TimeOfDay = 00:00:00;
                           Year = 2020;}, 22, 0.009082824313)]
```

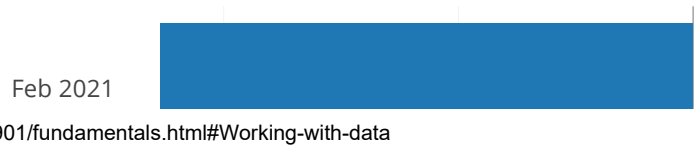
The default DateTime printing is too verbose if we don't care about time. We can simplify the printing:

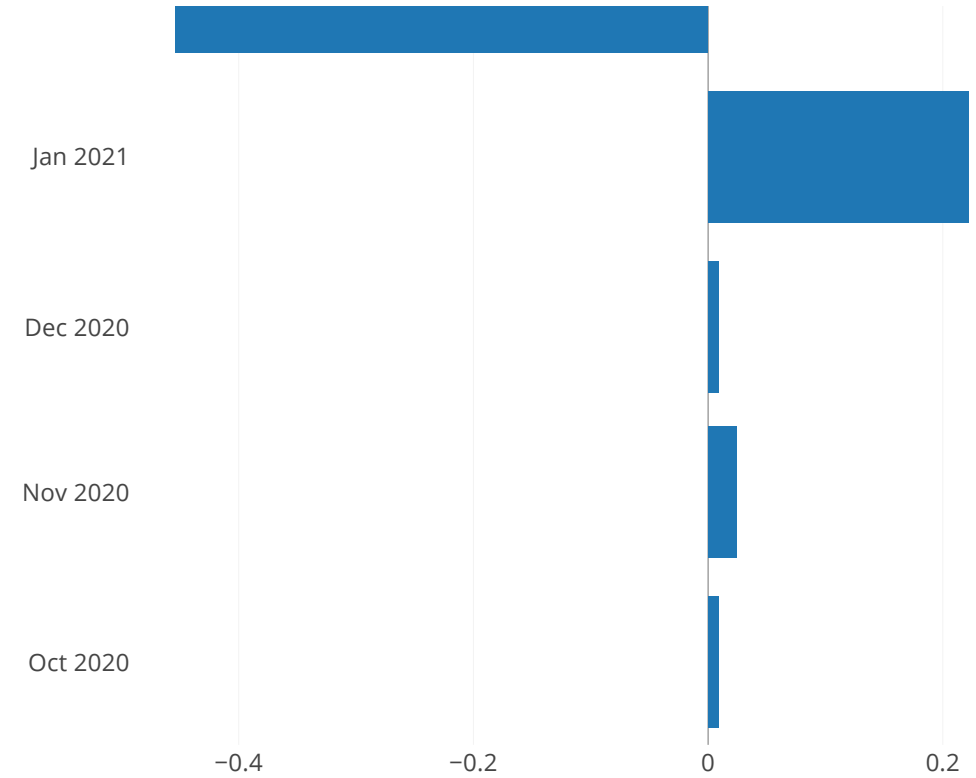
```
fsi.AddPrinter<DateTime>(fun dt -> dt.ToString("s"))
avgReturnEachMonth |> Seq.take 3 |> Seq.toList
```

```
val it : (DateTime * int * float) list =
  [(2020-10-01T00:00:00, 21, 0.008618813543);
   (2020-11-01T00:00:00, 20, 0.02443997661);
   (2020-12-01T00:00:00, 22, 0.009082824313)]
```

```
let monthlyReturnChart =
    avgReturnEachMonth
    |> Seq.map(fun (month, cnt, ret) -> month, ret)
    |> Chart.Bar
```

```
monthlyReturnChart |> Chart.Show
```





Volatility

We represent volatility by the standard deviation of returns. We can define a standard deviation function ourself

```
let stddev xs =
    let mu = xs |> Seq.average
    let sse = xs |> Seq.map(fun x -> (x - mu)**2.0) |> Seq.sum
    let n = xs |> Seq.length |> float
    sqrt (sse / (n - 1.0))

[1.0 .. 10.0 ] |> stddev
```

```
val stddev : xs:seq<float> -> float
val it : float = 3.027650354
```

But it is also convenient to use the FSharp.Stats

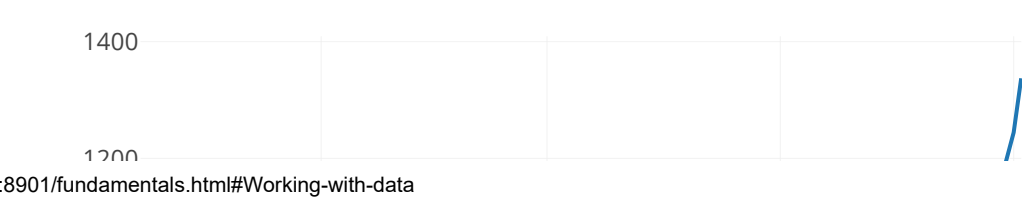
```
#r "nuget: FSharp.Stats, 0.4.0"

open FSharp.Stats
[1.0 .. 10.0 ] |> Seq.stDev
```

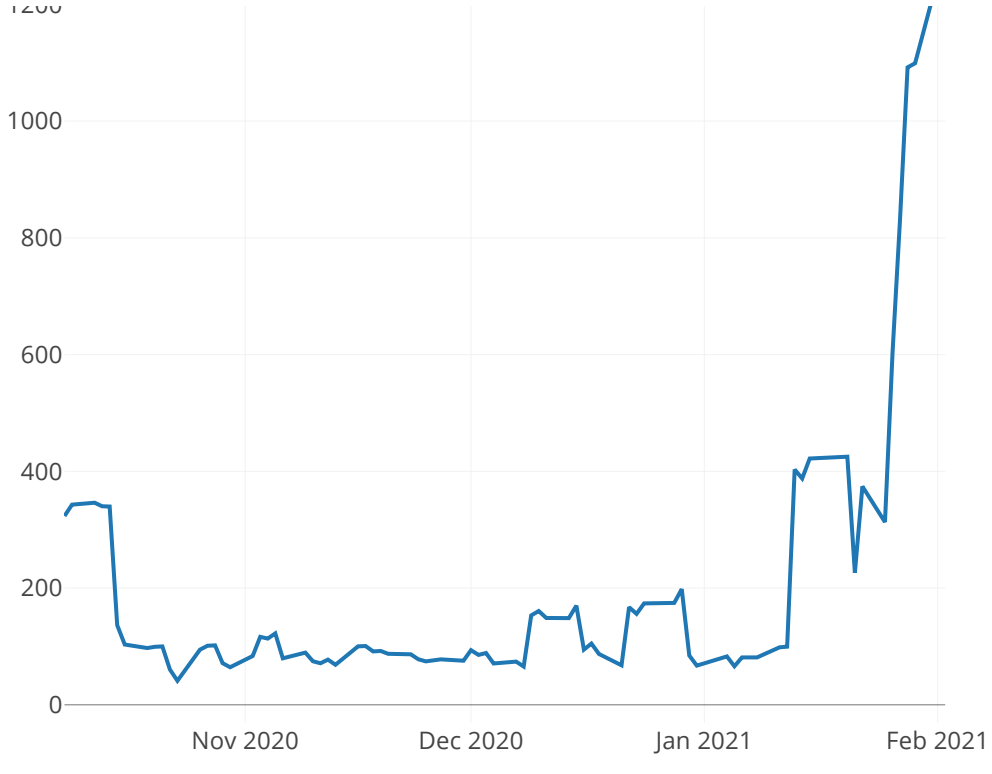
```
[Loading C:\Users\nicho\AppData\Local\Temp\nuget\44616--88164b2d-49cc-4061-a6ee-97d9ccb2aa4a\Project.fsproj.fsx]
namespace FSI_0232.Project

val it : float = 3.027650354
```

Now let's look at 5-day rolling volatilities.

```
let rollingVols =
    returns
    |> Seq.sortBy fst // never can be too careful
    |> Seq.windowed 5
    |> Seq.map(fun xs ->
        let maxWindowDate = xs |> Seq.map fst |> Seq.max
        let dailyVol = xs |> Seq.stDevBy snd
        let annualizedVolInPct = dailyVol * sqrt(252.0) * 100.0
        maxWindowDate, annualizedVolInPct)

let volChart =
    rollingVols
    |> Chart.Line
```

```
volChart |> Chart.Show
```




1000
800
600
400
200
0
Nov 2020
Dec 2020
Jan 2021
Feb 2021